University of Zurich[UZH]

Zurich Open Repository and Archive
University of Zurich
University Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2015

Intent, Tests, and Release Dependencies: Pragmatic Recipes for Source Code Integration

Brandtner, Martin ; Leitner, Philipp ; Gall, Harald C

DOI: https://doi.org/10.1109/SCAM.2015.7335397

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-112027
Conference or Workshop Item
Published Version

Originally published at:
Brandtner, Martin; Leitner, Philipp; Gall, Harald C (2015). Intent, Tests, and Release Dependencies: Pragmatic Recipes for Source Code Integration. In: Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Bremen, Germany, 27 September 2015 - 28 September 2015, s.n..
DOI: https://doi.org/10.1109/SCAM.2015.7335397

Intent, Tests, and Release Dependencies: Pragmatic Recipes for Source Code Integration

Martin Brandtner, Philipp Leitner, and Harald C. Gall
University of Zurich, Department of Informatics, Switzerland
Email: {brandtner, leitner, gall}@ifi.uzh.ch

***Abstract*—Continuous integration of source code changes, for example, via pull-request driven contribution channels, has become standard in many software projects. However, the decision to integrate source code changes into a release is complex and has to be taken by a software manager. In this work, we identify a set of three pragmatic recipes plus variations to support the decision making of integrating code contributions into a release. These recipes cover the isolation of source code changes, contribution of test code, and the linking of commits to issues. We analyze the development history of 21 open-source software projects, to evaluate whether, and to what extent, those recipes are followed in open-source projects. The results of our analysis showed that open-source projects largely follow recipes on a compliance level of $> 75\%$. Hence, we conclude that the identified recipes plus variations can be seen as wide-spread relevant best-practices for source code integration.**

I. Introduction

Over the last years, the importance of continuously integrating source code contributions has raised in software development. Pull-request-driven contribution channels (e.g., GitHub [1], [2]) open comfortable ways to submit contributions to software projects. Continuous integration encompasses various kinds of platforms, including (1) version control systems (e.g., Git), (2) issue tracking (e.g., JIRA), or (3) build and release management systems (e.g., Jenkins). These platforms, as well as the artifacts associated with them, are not independent.

In this paper, we focus on the intersection of source code contributions and issues during the integration process. Generally, there are two different classes of problems that software managers face related to this intersection. Firstly, **feature and release management** requires software managers to stay aware of which feature or bug fix is contained in which contribution. Further, release management also requires isolation of changes, so that, for example, single feature implementations can easily be integrated into specific releases. Secondly, **contribution quality management** requires software managers to make sure that source code contributions submitted for issues are of sufficient quality and actually implement the required change.

In practice, various conventions and guidelines have been established to support software managers with feature, release, and quality management. For example, to ease release management, the Apache Software Foundation (ASF) has established a guideline [3] that every commit message has to contain a reference to an entry in the issue tracker via the issue identifier. Similar guidelines exist in a plethora of other software projects and can help to avoid non optimal source code changes (e.g., [4], [5]). However, those guidelines are generally neither well-defined, nor portable between projects, nor are there tools to support their convenient monitoring.

In this paper, we propose a set of pragmatic guidelines, referred to as **recipes**, to foster the interlinking of software development artifacts and tools. Recipes are built on project-independent best-practices, and consist of a **rule** (the actual guideline), one or more **ingredients** (artifacts or data), optional **variations** (for project-specific requirements), and a **purpose** (why a recipe is useful). We interviewed software managers to capture their best-practices for the definition of an initial set of recipes. A software manager in the context of our work has one of the following roles: lead developer, senior developer, or senior software architect. In the second step, we analyzed the development history of 21 open-source projects from the ASF, JBoss, and Spring to see to what extent the proposed recipes can be identified in open-source projects. A manual monitoring of even simple recipes is extremely time-consuming. For example, the history of the Apache Ambari project contains 2577 commits from 39 different contributors between July 2014 and December 2014. In the same time period, 2450 issues were created or resolved. It is obvious that it is not feasible for a software manager to manually keep track of every single source code contribution or issue and its impact, especially if she is involved in multiple projects at the same time. Hence, we have devised semi-automated tooling to foster the inspection of existing projects for compliance with the proposed recipes.

The three main contributions of this paper are:

- The definition of **three pragmatic recipes plus variations** for source code integration based on best-practices in three software companies.
- A **quantitative analysis** of 21 projects from ASF, JBoss, and Spring to validate the relevance and use of these recipes in open-source software.
- A **qualitative analysis** based on the same projects, to determine reasons for cases in which recipes get violated.

In particular, we want to highlight three general findings, besides the specific ones described later in the paper: (1) for each source code contribution it is important to state its intent, tests, and dependencies, (2) open-source projects largely follow best-practices from industry, and (3) while compliance to these principles is generally substantial, there are cases where

/15/$31.00 © 2015 IEEE

deviations make sense from a software developer's or project manager's point of view.

The remainder of this paper is structured as follows. In Section II, we introduce our research design, followed by detailed description of the research process and its results in Sections III and IV. In Section V, we discuss the results followed by the most relevant related work in Section VI. The threats to validity of our research are summarized in Section VII. Finally, we conclude with our main findings in Section VIII.

II. Research Design

Our work addresses the following two research questions:

RQ1: *What recipes can support software managers during the integration of source code contributions?*

RQ2: *To which extent do open-source projects follow source code integration recipes, such as those proposed in this work?*

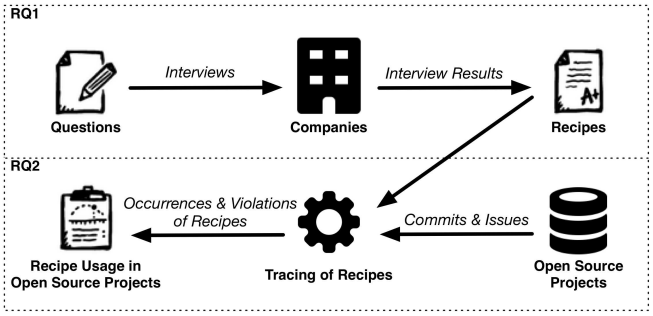

Fig. 1. Approach Overview

Our approach to address these questions follows a two-step research design, as illustrated in Figure 1. The first step is an interview of software managers to collect best-practices for source code integration. In a second step, we evaluate to what extent the proposed recipes can be identified in open-source projects, and in which situations open-source developers deviate from the proposed best-practices. For this step, we mined artifacts from 21 open-source projects to analyse their recipe compliance, and we qualitatively investigated cases of non-compliance.

III. Pragmatic Code Integration

In the first step of our research, we interviewed software managers from industry to discover best-practices for the integration of source code contributions into a code base.

A. Methodology

We contacted software development departments of seven international companies with headquarter or branches in Switzerland. Those companies were active within the banking, consulting, and software engineering sector. In a short summary, we presented the findings of our preliminary work [6] and asked them for participation. Three companies volunteered to participate in our interviews, and we scheduled interview sessions with three to four employees per company. The duration of an interview session per participant was 30 minutes, and the sessions took place at the interview partner's offices. At the beginning of each session, we collected standard background information on the participant (see Section III-B). We asked each participant for best-practices that are established in her company, and how she judges its importance for source code integration. Each interview session is documented with an audio record and semi-structured notes in a questionnaire style. For the design of the interview sessions we followed the guidelines of Kitchenham and Pfleeger [7]. The questionnaire was filled out by the interviewer in accordance with the interview partner.

B. Interview Partners

We interviewed eleven people from software development departments of three different international companies located in Switzerland. The average software development experience of our interview partners was 12.3 years. Each interview partner has a solid background in developing Java applications, and knows about the challenges of integrating source code contributions.

TABLE I
Summary of interview partner's job titles and professional backgrounds

	Job Title	Experience	Company
P1	Software Manager	23 years	A
P2	Software Manager	10 years	A
P3	Software Manager	20 years	A
P4	Software Manager	8 years	A
P5	Lead-Developer	10 years	B
P6	Senior Developer	14 years	B
P7	Lead-Developer	12 years	B
P8	Senior Developer	11 years	C
P9	Senior Developer	10 years	C
P10	Lead-Developer	10 years	C
P11	Software Manager	7 years	C

Half of the interview partners work in multiple projects concurrently. None of the interview partners worked in the same project as one of the other interview partners. Table I lists the self-described job title, the software development experience, and the company affiliation of our interview partners.

C. Interview Results

In the following, we list the resulting recipes R1-R3. Each recipe consists of a rule (the actual guideline), ingredients (artifacts or data), variations (for project-specific requirements), a purpose (why a recipe is useful), and a description based on the results of the interviews.

R1 – Reveal the Intent.

Our interview partners rated this recipe at least as *important* to keep track of what concrete bug or change request a code contribution actually solves. This eases the task of deciding whether a given contribution should be part of a new release. Some participants shared a less restrictive perspective on this topic. They claimed that the effort of creating a new issue entry even for small source code change is too large:

Reveal the Intent

Rule:	Contributions should contain at least one link to an issue
Ingredients:	Commit message, kind of commit (e.g., code, merge, etc.), number of changed resources per commit
Variation:	R1.1: *Small* contributions should contain *either* a link to an issue *or* a useful description
Purpose:	*Release management* – documenting which issue a source code contribution fixes

> "*For smaller commits, the creation of an issue may last too long. In such a case, a description of the problem and the solution directly in the commit message is preferred.*" –P4

Hence, we have established a variation **R1.1** of the basic recipe to cover the exceptional case of small commits. All interview partners used the number of changed resources to define commits that qualify as "small", but we were not able to find a common definition as the mentioned values vary between one and up to ten changed resources, depending on the changed resource type. The ingredients of this recipe are the commit data from the VCS. Depending on the project, the interpretation of this recipe varies in the definition of a small commit, and the reference key or description style used in the commit message.

R2 – Align the Tests.

Align the Tests

Rule:	Changes in source and test code should be combined in one commit
Ingredients:	Ratio of source and test code changes in a commit
Variation:	R2.1: Commits that contain only source code changes should be proceeded or followed by a commit containing test code changes
Purpose:	*Quality management* – ensuring testing of source code contributions

This recipe states that source code contributions and tests should be commited together. This is rated as important or very important, depending on the degree of existing test coverage within a software system:

> "*Such a recipe is only important for new projects and existing projects with a good test coverage.*" –P2

One reason mentioned multiple times is the effort of the preparation work, which would be needed to create meaningful test cases in a project that is currently not widely using a test environment. For example, the creation of a test case, which relies on data from a database, requires mocks [8] or a testing database. In such cases, the preparation effort is higher than the actual code change in a software system. A further impact on this recipe raised by our interview partners are company-specific modifications of test-driven development. For example, in some projects the test code changes get committed before the actual source code change in case of a bugfix but not in case of a new feature.

> "*It is important to ensure the contribution of test code, but not every commit addressing an issue contains source code and test code changes.*" –P9

We established a variation **R2.1** of the basic recipe to cover situations in which source code and the test code changes get committed in different commits. The ingredient of this recipe is the commit data, especially the list of changed resources from the VCS. Depending on the test setup in a software project, the interpretation of this recipe varies on the ratio between source code changes and test code changes within one commit. For example, in a project with a high test coverage, a change in the source code might led to a relatively small change of the test code, whereas in a project with a low test coverage, a small source code change may lead to a large change in the test code.

R3 – Reveal the Release Dependencies.

Reveal the Release Dependencies

Rule:	Commits that address multiple issues should be marked as such
Ingredients:	Commit message Keys of linked/duplicated issues
Variation:	None
Purpose:	*Release management* – isolation of source code changes

The isolation of source code changes, addressed in this recipe, is important as it allows software managers to cherry-pick different feature commits and bugfixes for a release. Some interview partners address the isolation of source code changes by allowing only one issue key per commit message.

> "*We enforce commit messages with only one issue key and use dummy issues to encourage developers to not include issue independent changes (e.g., typos, etc.) in a commit.*" –P5

Others allow multiple issue keys per commit message in case that a commit addresses multiple issues. Alternatively, a developer may reference one issue in the commit message, and mark the others as duplicates of the referenced one.

The ingredients of this recipe are the commit data from the VCS and the data of resolved issues from the issue tracker. Depending on the project, duplicated or related issues get explicitly linked in the issue tracker, and have multiple issue keys in the commit message. For example, in case of a resolved issue without a commit in the VCS (issue key is never mentioned in a commit message) a link to another issue should exist in the issue tracker.

IV. Analysis and Results

In the second step of our study, we validated those recipes based on open-source project data. We mined projects from the ASF, JBoss, and Spring in order to trace occurrences of the proposed recipes in the wild. We extracted data of a six months period (July 2014 to December 2014) from the version control system (VCS) and the issue tracker hosted by the projects.

A. Project Selection and Data Extraction

We investigated projects from well-known open-source communities, such as ASF, Eclipse, JBoss, or Mozilla. Initial analysis showed that most of the projects of the mentioned communities use Git as VCS, or at least offer a Git mirror of the SVN repository. JIRA is used as issue tracker by most of the projects within the ASF and JBoss community, and Bugzilla is used by most of the projects within the Eclipse and Mozilla community. Furthermore, we looked at the used programming language and the build automation systems used within the different open-source projects. We decided to analyze Java-based open-source projects from the ASF, JBoss, and Spring community, primarily due to their active development history. We used the platform OpenHUB.net to search for Java projects from the ASF, JBoss, and Spring community under active development. Based on this search, we randomly selected 60 Java projects that use Maven from the ASF (40), the JBoss (10), and the Spring (10) community. We discarded project with less than 120 created or resolved issues, and less than 120 commits in the analyzed time-period of six months (on average 20 commits/issues per month).

TABLE II
Development activity overview of the 21 selected projects (July 2014 to December 2014)

Project	Commits	Issues	Committers
Apache			
Accumulo	630	403	17
ActiveMQ	271	179	12
Ambari	2577	2450	39
Camel	1669	591	44
CXF	805	294	16
Drill	435	611	13
Felix	353	200	17
HBase	998	1374	29
Hive	1310	1663	25
J.rabbit-Oak	801	360	15
Karaf	272	354	9
PDFBox	936	616	6
Sling	1512	473	20
Spark	1617	2158	19
TomEE	510	184	6
JBoss			
Richfaces	388	329	12
WildFly	591	560	75
Windup	368	231	8
Spring			
Framework	942	703	20
Integration	165	167	6
XD	426	437	24

The remaining 21 projects are listed in Table II. One of the largest projects in terms of development activities is the Apache Ambari project, with 2577 commits from 39 contributors and 2450 issue changes (created/resolved). Based on these numbers, a committer of the Apache Ambari project contributed on average 66.08 change sets in the second half of 2014. We developed a Java application to extract the source code history and issue data used for our further analysis. After extraction, we stored the data in a relational database. For the extraction of the Git data, we used the JGit[1] library, and the issue tracker data was extracted through the REST API of JIRA. An archive with the resulting data is available for download on our website.[2]

B. Recipes in Open-Source Projects

We use the extracted data for the further compliance analysis in this paper, which consists of a quantitative and qualitative discussion for each recipe.

1) Reveal the Intent: For the analysis of this recipe, we separate commits into two groups, *small commits* and *large commits*. In our context, we define commits as "small" if they address only one source code file and as "large" if they address multiple source code files. According to the interview results, small commits do not necessarily require an issue key in the commit message, but should have a description.

Quantitative analysis. We analyzed the commits of each group and looked for commit messages containing issue keys based on the JIRA issue key identifiers of each project. For example, the issue identifier for the Apache Camel project is *CAMEL*, and the format of an issue key (i.e., issue #1332) is as follows: *CAMEL-1332*. Additionally, we calculated the average commit message length within each commit group to find out if the absence of issue keys influences the length of the provided description in the commit message.

Result R1: 14 of 21 projects use issue keys in more than 75% of large commits.

	0-50%	51-75%	76-90%	91-100%
# of Projects	1	6	6	8

Two-thirds of the analyzed projects regularly use issue keys in commit messages of changes addressing more than one source code file. In the remaining third of the projects, we also found large commits with issue key. However, in these cases the issue keys were not used regularly (<76% with issue key). Especially in case of the Apache TomEE project, the usage of issues keys for large commits is not overly established (only 26% of contributions contain an issue key).

Result R1.1: 9 of 21 projects use issue keys in more than 75% of small commits.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	7	5	4	5

The results are different in case of small commits, 57% of the analyzed projects do not use issue keys for those commits

[1]JGit-http://www.eclipse.org/jgit/

[2]http://www.ifi.uzh.ch/seal/people/brandtner/projects/recipes.html

on a regular basis (<76%). This is also reflected in Figure 2, which depicts the usage of issue keys in commit messages for the different change set sizes calculated across all analyzed projects and for a selection of projects.

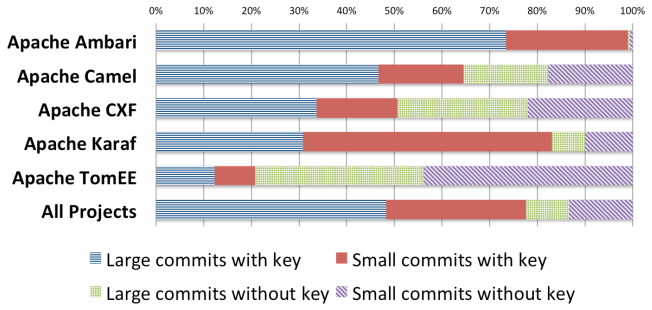

Fig. 2. Commits with and without an issue key in the commit message

In respect to variation R1.1 of recipe R1, we analyzed also the average commit message length. The analysis showed that the average message length of commits without an issue key is up to 69% shorter (e.g., Apache Felix) than those of commits with issue key. Only in case of the Apache Ambari project, the commit message of small commits without an issue key was on average 12% longer compared to the message of small commits with issue key.

Qualitative analysis. In order to conduct a further qualitative analysis of cases when issue keys have been omitted, we decided to investigate a randomly selected subset of five commits per project without key. We filtered source code commits without issue key and categorized the resulting 42 commits according the described change in the commit message. The categories are: build failure (11.9%), dependency management (19%), rollback (7.1%), versioning (35.7%), and other changes (26.3%).

Build failure. Problems related to the build process of a software system are often mentioned in commit messages without an issue key. For example, in the Apache Accumulo project: *"unbreaking build; trivial change"* or the Apache PDFBox project: *"added rat exclude rules to avoid build failures"*. As indicated by the commit messages, those commits usually not not contain large changes.

Dependency management. Version upgrades of used libraries or similar changes are potential candidates for commits without an issue key. For example, in the Apache Jackrabbit project: *"use latest H2 version [...]"*. Those kind of commits often only consist of changes to the *pom.xml* files of Maven-based software projects.

Rollback. Commits that are completely or partly reverted to a previous state are candidates for commits without an issue key. For example, in the Apache Hive project: *"Rolled back to 1643551"*. As shown in the example, the message of a rollback commit often contains only a reference in the VCS but no reference to the affected issue in the issue tracker.

Versioning. The release of a software system includes the change of the according versioning information in the software artifacts. Some of the analyzed projects (e.g., Apache Felix, or Apache Sling) use the Maven release-plugin for this step. The default commit message generated by the Maven release-plugin starts with *"[maven-release-plugin]"* and does not contain an issue key.

Other changes. Other changes that often do not contain links to issues are the polishing of code snippets, the addition of missing files, and typo fixes.

Summary. We were able to show that most of the analyzed projects (66.7%) comply with recipe R1 to a compliance level of at least 75%. However, each of these projects has a residual of commits that deviate from this recipe.

Exceptions. Fixing build failures, dependency management, rollbacks, or versioning are exceptions that cause violations of this recipe. Those changes are not directly linked to any concrete issue, and hence can do without explicit link. Most importantly, missing links of these changes are typically not detrimental to the stated purpose of this recipe (release management).

2) Align the Tests: For this analysis, we established four groups for the classification of commits based on the affected resources: (1) source code files, (2) test code files, (3) source code and test code files (combined commits), and (4) other files. This classification is inspired by the structure of the Maven-based projects used in our analysis. Any change in a resource located under *src/main* is classified as *source code* change. A *test code* change is any change of a resource located under *src/test*. Every resource changes outside one of the two mentioned locations is classified as *other* change. For the association of source and test code changes, we depended on the naming schema of Maven. The test code for a class in the *src/main* directory has to resist under the same relative path in the *src/test* directory with a *Test* suffix in the resource name. For example, the test code for a class *src/main/ClassA.java* is located under *test/main/ClassATest.java*.

Quantitative analysis. We analyzed the structure of source code, test code, and combined commits of each project. As many of the analyzed projects are organized through modules, we had to take care of cascading directory structures, which means multiple *src/main* and *src/test* directories within a repository.

Result R2: In 1 of 21 projects more than 75% of the commits contain source code and test code changes.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	17	3	1	0

The results of the analysis showed that only in case of the Spring Integration project, the majority of the source code changes (79%) get committed together with test cases. An average value of 30.7% (across all projects) indicates that combined source code and test code commits do exist, but they are not the largest group of commits existing in open-source projects. The largest group of commits are pure source code commits with an average value of 53.4% (across all projects).

The group of pure test code commits reach an average value of 15.9% (across all projects).

Result R2.1: In 14 of 21 projects more than 75% of the source code commits have test code commits.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	3	4	10	4

In many cases, source code and test code changes end up in the VCS as independent commits. Addressing the amount of independent source code and test code commits, the compliance is better compared to the compliance of the combined commits. 14 of 21 of the analyzed projects have a similar amount of pure source code and test code commits resulting in a high compliance level (>75%). For example, the Spring Integration project in Figure 3.

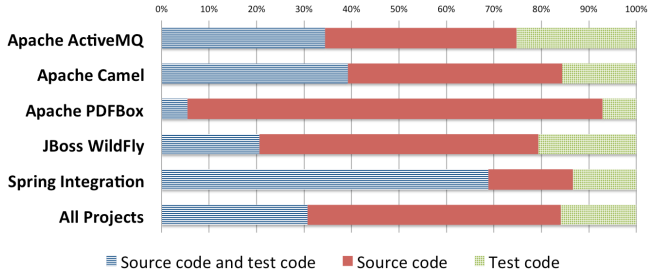

Fig. 3. Share of resources affected by commits

However, the shares computed across all projects indicate the there are projects which do not follow this recipe. One example for such a project is Apache PDFBox. The majority of commits (87%) in the VCS of this projects are pure source code changes and only 5% of the commits are combined ones. Such a large amount of pure source code commits can be an indication for shortcomings in the testing process.

Qualitative analysis. In order to dig deeper into why source and test code is not commited together, we selected all issues that have at least one comment containing the term *"unit test"*. Most of the comments in the resulting 1632 issues do not address the missing testing of a contribution. Therefore, we manually inspected 50 issues and found three issues that are affected by this recipe.

The found issues indicate that the management of open-source software projects take care of test code contributions through patches. For example, a patch for issue SLING-4212 of the Apache Sling project was not integrated and commented with: *"[...] add unit tests for the case name = null"*. A similar comment can be found for issue SLING-4112: *"open points: review if we should add some more specific unit tests for the new classes [...]"*. Again, the according source code was not directly integrated into the codebase. A slightly different example was found in the Apache Jackrabbit-Oak project (OAK-2301): *"[...] That needs to be done, plus unit tests."* Despite this comment, the patch was integrated into the codebase without any tests. Such a behavior can be an indicator for the performance differences between recipe R2 and its variation R2.1 in the qualitative analysis, as test code changes get integrated at another point in time as the according source code changes (or not at all).

Summary. We were able to show that many of the projects (66.7%) comply at least with R2.1, a variation of recipe R2. However, depending on the project, this recipe sometimes gets violated or is not followed at all.
Exceptions. A typical exception of this recipe is the integration of a patch without taking care of missing tests.

3) Reveal the Release Dependencies: In this analysis, we focused on issues of the types *bug* or *feature* exclusively, as we expect a commit in the VCS for these types. The resulting set of issues was split into three groups: (1) issues addressed in an exclusive commit, (2) issues addressed in bulk commits, and (3) issues without commit. An exclusive commit in this context is manifested through a commit messages that contains exactly one issue key and an optional description. A bulk commit is defined in our context as commit that contains multiple issue keys in the commit message.

Quantitative analysis. We analyzed the structure of issues that have been marked as resolved based on changes in a bulk commit and resolved issues without commit.

Result: In all projects more than 75% of the commits that address multiple issues were marked as such.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	0	0	7	14

The results showed that commits addressing multiple issues at once exist in all of the analyzed projects. Each project takes care of marking such commits with an issue key for every affected issue.

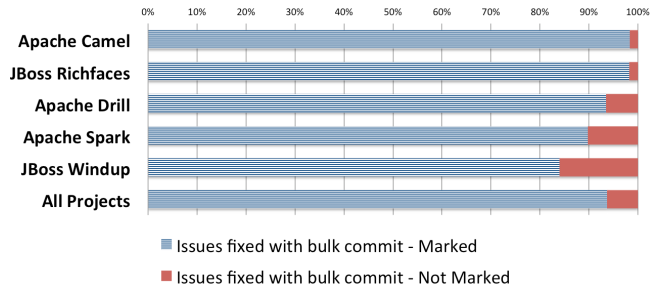

Fig. 4. Resolved issues with and without dedicated commits

Figure 4 depicts the issue shares calculated across all projects and a selection of five further projects. The overall share of issues fixed with a bulk commit are marked as such on average in 93.7% of the cases. This can be seen as a strong indication for the existence and the enforcement of this recipe in open-source projects.

Qualitative analysis. For the qualitative analysis, we decided to investigate the rare cases in which an issue was marked as resolved without an associated source code change.

We manually investigated a subset of 50 resolved issues without an associated commit. In this subset, we found five issues that violate recipe R3. One of these issues is FELIX-4654, which seems to be resolved by the code change of another issue. However, there is only a code snippet of the latest development snapshot mentioned in the comment. Based on the information in the issue tracker, it is not possible to determine the related issue or commit. A similar case is issue AMQ-5313 of the Apache ActiveMQ project. This issue was also fixed by a change that addresses another issue. Instead of linking the according issue tracker entry, the developer describes the changed configuration and its effects in the issue comments. The result is an issue entry with plenty of text that is hard to understand for all involved stakeholders.

Another interesting case is issue KARAF-3218. This issue is marked as duplicate of issue KARAF-3075, which addresses the underlying problem that causes both issues. However, the difference of issue KARAF-3075 and KARAF-3218 is the affected version of Karaf. In KARAF-3218, the issue is raised for version 2.3.2 and in KARAF-3075 it is raised for version 2.3.5 and later. To solve issue KARAF-3075, the code change was initially applied to the branches of version 2.4.x and later. The situation was clarified with issue KARAF-3417, where the reporter of issue KARAF-3218 asked for a backport.

Summary. We were able to show that all of the projects comply with recipe R3 to a compliance level of 75% or more. However, each of these projects has a small residual (on average $<7\%$) of commits that violate this recipe.
Exceptions. Issue entries describing side-effects caused by other issues are exceptions that violate this recipe. Such entries have in common that no commit is associated to them, but the comments contain clear indications that the issue disappeared with a code change that affects another issue.

V. Discussion

The results of our interview study in industry as well as the quantitative and qualitative analysis of open-source projects indicate that software projects largely follow the proposed recipes for release and quality management. Table III summarizes the pragmatic recipes, the median compliance level, and the number of projects that fulfill a certain recipe on a *compliance level of 75% or more*.

In the following, we discuss the analysis results of the proposed recipes according to their purpose.

Feature and release management. Two of the proposed recipes ("R1 – Reveal the Intent" and "R3 – Reveal the Release Dependencies") address the feature and release management of the source code integration process. For both recipes, the declaration of the commit content and the correct association to an issue plays an important role.

Especially before code freeze deadlines, software managers have to handle large amounts of source code contributions. It is impossible for them to inspect the content and the purpose of each single source code contribution. Time constraints caused by, for example, release plans force software managers to decide whether or not to integrate a contribution within a few minutes. A clear declaration of a contribution's content with an issue tracker entry can support the decision making and reduce the risk of integrating less important or even unneeded changes into a code base.

The interviews as well as the analysis showed that projects follow these recipes to a large extent. Despite the rather high compliance rate of the analyzed projects, it is still possible to find violations for each of the recipes. A typical reason that we found for such a violation is the contribution of source code that addresses a small change (e.g., typo fix). The violations which we found in the open-source projects are in line with the interview results of allowing small code changes without issue entry. During the interviews, the relatively high effort of opening issues was mentioned as a hurdle to having issue keys even for small changes. However, it remains unclear whether the effort saved when implementing a source code change without issue key is more substantial than the additional effort caused by the absence of a clear purpose during the feature integration, especially considering that the time of a software manager is often more costly than that of a developer. Of course, there are cases in which the creation of an issue entry can indeed be seen as overkill. For example, a version change of a dependency in a Maven-based software project is achieved by simply replacing the old version number through the new one in the *pom.xml* file. Creating an issue entry for every such case is usually not important, especially given that the implications of such changes for feature integration are limited.

Exceptional cases are source code contributions that address multiple issues at once. The analysis showed that those contributions contain a reference to at least one issue entry, but there is no guarantee that each affected issue is indeed revealed. The absence of a reference to an issue entry that is addressed by a commit can be caused by various reasons. In our analysis, we were able to find at least two different cases in which an issue was not referenced by the issue resolving commit. In one case, the issue seemed to be caused by a side-effect of another issue. As soon as the underlying issue was resolved, the side-effect disappeared. However, we were not able to verify the potential issue relationship based on the data available in the VCS and the issue tracker. Especially the textual description of potential relationships mentioned in the comments of the according issues without mentioning an issue number makes it impossible to verify relationships. In another case, the wrong classification of an issue as duplicate of an existing one led to a situation in which a bug fix was applied to a subset of affected versions only. Therefore, the absence of the issue reference in the commit message was correct even if the commit contains the actual fix. In this case the failure occurred on the issue management level and not on the source code integration level.

Contribution quality management. One of the proposed recipes ("R2 – Align the Tests") addresses the internal quality

TABLE III
MEDIAN COMPLIANCE LEVEL AND NUMBER OF PROJECTS WITH A COMPLIANCE LEVEL GREATER THAN 75%

	Recipe	Median	>75%
R1	Contributions should contain at least one link to an issue	81.6%	14
R1.1	Small contributions should contain either a link to an issue or a useful description	57.2%	9
R2	Changes in source and test code should be combined in one commit	40.2%	1
R2.1	Commits that contain only source code changes should proceed or followed by a commit containing test code changes	77.8%	14
R3	Commits that solve multiple issues should be marked as such	93.7%	21

of source code contributions and its impact on the integration process.

An important goal of software managers is to get new features implemented and bugs fixed within an appropriate time span. This time span varies depending on the project and the priority of the underlying issue. An important attribute in such a scenario is the quality testing of a source code contribution to ensure the expected behavior of a software system after integration. The interviews as well as the analysis showed that many projects follow this recipe, but the level of compliance is strongly influenced by the history of a software project. For example, it is more likely that a software project, which followed this recipe from the beginning, continue following this recipe during the remaining development life cycle. We were not able to find a project that started off with no or very low test coverage, but started to strictly follow R2 at a later phase of the project. Those findings are in line with the results of the interview study. All of our interview partners confirmed that starting following such a recipe in a project only makes sense at the beginning. The major challenge in all of the projects was to reach a solid testing density across the whole system. A high density of tests for single modules or classes is nice to have, but does not necessarily reduce the risk of problems at the system at-large. For example, a high testing density of a module that represents one percent of the whole code base cannot reduce the risk of a problem in the remaining ninety-nine percent of the code. Of course, this also depends on the criticality and error-proneness of the module in a software system. It is likely that a high test density for a central and error-prone module can reduce the risk of a failure more efficiently than a high test density of a rarely used utility module.

Our analysis showed that only in a few cases, all projects violate variation R2.1, but such violations occur less often than for R2. We assume that a violation of this recipe is strongly correlated to the importance and the size of a code change. For example, we found cases where software managers commented that the code change was integrated without code changes, but the issue remains open as long as there is no test case contributed. Most according issues were classified as bugs and had at least a *major* priority, supporting our theory that this is particularly prone to happen for high-impact bugs for which the software manager quickly needs a fix and is willing to live with quality risks to have the problem solved. In a small number of other cases, software managers integrated a source code change and wrote a comment that the tests are missing, but nevertheless marked the issues as resolved. The according issue entries for these cases were often classified as minor improvements with a low priority.

In a further investigation, we found indicators that the number of test code changes indeed impacts the test coverage of a software project. For example, the quantitative analysis results of the Apache PDFBox project indicate a rather low number of changes in the project's test code. A public available quality analysis of the latest PDFBox version[3] showed that the low number of test code changes is reflected in a rather low test coverage (18.7%) as well.

Finally, as a corollary of our study, we found that the *Hadoop QA* bot is used in multiple of the analyzed projects (e.g., Ambari, HBase, Hive). This bot automatically conducts a number of heuristic quality checks for every submitted source code contribution, two of which concern testing. The first such check is the number of new or modified tests (e.g.,*"+1 tests included. The patch appears to include 3 new or modified test files"*), the second check addresses the test results (e.g., *"+1 core tests. The patch passed unit tests in ambari-server."*). We speculate that such a bot can be beneficial, especially for the decision making of a software manager during the integration process. For example, for a quick check if test cases have been changed, it is no longer necessary to dig into the source code of every commit with such tooling.

VI. RELATED WORK

The integration of source code contributions and issue management are often discussed topics in literature. We divide the field of the related work into two areas: source code contribution management and issue management.

Source code contribution management: Tsay et al. [2] investigated source code contributions in pull-request-based environments, such as GitHub. They analyzed different aspects for the rejection or the integration of source code contributions into the main code base. The results of their research showed that the integration of smaller contributions is more likely compared to large source code changes affecting multiple files. Pham et al. [9] investigated the test culture in social coding platforms and found out that the inclusion of test cases can raise the likelihood for integration as well. Gousios et al. [1] additionally investigated issue descriptions in social coding

[3]http://nemo.sonarqube.org/dashboard/index/332186 [accessed: Jan 26, 2015]

platforms, and showed that an insufficient task articulation is the major rejection reason for source code contributions. The research on branching in software projects addresses source code integration as well. However, branching is a different paradigm, which supplements the pull-request-based paradigm. The investigation of related work on branching addresses aspects, such as reasons for the creation of a branch, the optimal points in time for the merge of a branch [10], and the impact of structural characteristics within a branch onto the source code quality after a merge [11]. A third way to contribute source code to a software projects is the submission of patches via a mailing list or an issue tracker. Research on the handling of patches addresses the reviewing and patch integration process in open-source projects, such as the Apache Server project (e.g., [12]) or the Linux Kernel (e.g., [13]). The investigations of Brun et al. [14] showed that, for example, a high number of changes addressed within a patch negatively influences the integration process.

Issue management: Anvik et al. [15] investigated an approach for a semi-automatic bug assignment based on a machine learning algorithm. They used the issue assignee-field and status-change data from the issue tracker of the Eclipse and Firefox project for their predictions. The problem they faced is that every project tends to use these fields differently (e.g., dummy assignee instead of a developer for each new bug, etc). Guo et al. [16] carried out a study in combination with a quantitative analysis to determine reasons for bug reassignments. A similar work with a focus on re-opened bugs was performed by Zimmermann et al. [17]. Their results showed that the initial priority assignment and poor bug descriptions strongly influence the number of reassignments and reopening of issues. The issue assignment process can be supported by results from investigations on the classification of issues. Rastkar et al. [18] proposed an bug summary generator to support developers to quickly access experiences of other developers within the same project, which can give an insight for the issue assignment as well. Giger et al. [19] proposed a prediction model to estimate the expected time needed to resolve a certain bug. This prediction may help with issue prioritization. However, the issue assignment process and the proposed predictions preferable work with complete bug descriptions. Aranda and Venolia [20] investigated coordination patterns in issue trackers during bug fixing. One finding of their work is that bulk data changes in an issue tracker (e.g., status change of multiple issues) may negatively influence coordination patterns. Bettenburg et al. [21] conducted a study with developers and bug reporters to find out what information the developers expect in a bug report to measure the quality of a bug. They found out that there is often a mismatch between the provided information of users and the information needed by developers. These results align with a study of Ko and Chilana [22], which concludes that the major impact of Mozilla's public issue tracker onto the development process are not the bug reports, but the hiring of talented developers. According to the work of Breu et al. [23], it is important to keep issues up-to-date with, for example, comments. Such status updates can effectively engage all involved stakeholders and support the project management.

To the best of our knowledge, there is no work that investigates a pragmatic source code integration approach for open-source projects based on best-practices from industry.

VII. Threats to Validity

Interviews and empirical studies have limitations that have to be considered when interpreting the results of our work.

Threats to external validity. These threats address the ability to generalize our results for software projects. The relatively low number of interview partners limits the ability to generalize the proposed set of recipes for software projects across different domains or companies. Further interviews with interview partners from other companies or domains may help to overcome this limitation. In our analysis, we used merged data from Java-based open-source projects of three open-source communities. We ignored differences in the project maturity and size during the analysis. Therefore, projects with more development activity may influence the average results more than projects with low activity. Furthermore, the results of our analysis are restricted to software projects that use Java as programming language, JIRA as issue tracker, and Maven as build system.

Threats to internal validity. These threats address the ability to draw conclusions based on our interview results. The use of an example set of draft recipes may introduce a bias in the interview results. We assume that the bias does not affect the results of the importance rating, as it reflects the personal opinion of each interview partner about the proposed statements. Another threat is the potential survival bias caused by the use of data mined from the VCS and the issue tracker. Especially the data of the VCS represents only cases of source code contribution that have been accepted but misses those, which have been rejected. We tried to mitigate this threat by focusing on patch contributions uploaded to the issue tracker. In difference to the VCS, the issue tracker stores the whole history of an issue and the contributed patches, independent of the decision to accept or reject it.

VIII. Conclusions

The analysis of 21 open-source projects showed that the integration of source code contributions in software projects largely follows recipes for feature, release, and quality management.

In this work, we propose an initial set of three pragmatic recipes plus variations for source code integration based on best-practices of software managers from industry. Our proposed recipes cover different aspects of Continuous Integration, from feature and release management to quality management. The proposed pragmatic recipes do not influence the technical process to integrate source code (e.g., merging, building, etc). Instead, these recipes enable a software manager to, for example, find source code contributions, which are not ready to be included into a release. Further, we analyzed to what extent open-source projects also follow similar recipes.

After studying 21 projects, we concluded that most of the identified recipes are wide-spread in the open-source community as well. The implications of our work for researchers and software managers are:

- Raise the awareness about the importance of the source code integration step.
- Reveal the intent, tests, or dependencies of a source code contribution to save value working time.
- Record the measurements and compliance (e.g., test coverage, code style) of each source code contribution.

Those implications can be directly applied in modern build and release management systems. For example, in some projects we found a lightweight approach called *HadoopQA* bot, which addresses some aspects of the listed implications.

Further, we need to explore more recipes for source code integration, both in industry and the open-source community. These additional recipes should not be limited to source code contributions and issues, but should also include other Continuous Integration data sources and artifacts. Our ultimate goal is to provide a collection of established best-practices, easing the adopting of Continuous Integration in practice.

REFERENCES

[1] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 345–355. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568260

[2] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 356–366. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568315

[3] Apache Software Foundation, "How should i apply patches from a contributor?" [Online]. Available: http://www.apache.org/dev/committers.html

[4] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1074–1083. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337363

[5] L. An, F. Khomh, and B. Adams, "Supplementary bug fixes vs. reopened bugs," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, Sept 2014, pp. 205–214.

[6] M. Brandtner, E. Giger, and H. Gall, "Supporting continuous integration by mashing-up software quality information," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, Feb. 2014, pp. 184–193.

[7] B. Kitchenham and S. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*. Springer London, 2008.

[8] K. Taneja, Y. Zhang, and T. Xie, "MODA: Automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 289–292. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859053

[9] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 112–121. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486804

[10] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 45:1–45:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393648

[11] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '12. New York, NY, USA: ACM, 2012, pp. 301–310. [Online]. Available: http://doi.acm.org/10.1145/2372251.2372305

[12] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: The apache server," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/337180.337209

[13] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: Case study on the linux kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 101–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487111

[14] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139

[15] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[16] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, ""not my bug!" and other reasons for software bug report reassignments," in *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, ser. CSCW '11. New York, NY, USA: ACM, 2011, pp. 395–404. [Online]. Available: http://doi.acm.org/10.1145/1958824.1958887

[17] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1074–1083. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337363

[18] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 505–514. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806872

[19] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808933

[20] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 298–308. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070530

[21] N. Bettenburg, S. Just, A. Schrter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 308–318. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453146

[22] A. J. Ko and P. K. Chilana, "How power users help and hinder open bug reporting," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1665–1674. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753576

[23] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: Improving cooperation between developers and users," in *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '10. New York, NY, USA: ACM, 2010, pp. 301–310. [Online]. Available: http://doi.acm.org/10.1145/1718918.1718973